|  |
| --- |
| Queueing Simulation |
| Tatar MaraAssignment no 3April 22, 2016 |

# Abstract

Objective

Design and implement a simulation application aiming to analyze queuing based systems for

determining and minimizing customers waiting time.

Description

Queues are commonly seen both in real world and in the models. The main objective of a

queue is to provide a place for a "customer" to wait before receiving a "service". The

management of queue based systems is interested in minimizing the time amount its

"customers" are waiting in queues. One way to minimize the waiting time is to add more

servers, i.e. more queues in the system (each queue is considered as having an associated

processor) but this approach increases the costs of the supplier. When a new server is added

the waiting customers will be evenly distributed to all current available queues.

The system should simulate a series of customers arriving for service, entering queues,

waiting, being served and finally leaving the queue. It tracks the time the customers spend

waiting in queues and outputs the average waiting time. To calculate waiting time we need to know the arrival time, finish time and service time. The arrival time and the service time

depend on the individual customers – when they show up and how much service they need.

The finish time depends on the number of queues, the number of other customers in the

queue and the service needs of those other customers.

Input data:

- Minimum and maximum interval of arriving time between customers;

- Minimum and maximum service time;

- Number of queues;

- Simulation interval;

- Other information you may consider necessary;

Minimal output:

- Average of waiting time, service time and empty queue time for 1, 2 and 3 queues for the

simulation interval and for a specified interval;

- Log of events and main system data

- Queue evolution

- Peak hour for the simulation interval

# Problem analysis ,use cases,scenarios

# Problem Analysis

Our goal is to implement an application that simulates a queuing system, that’s present in real life everywhere (at every shop, bank, paying point, etc.). The idea is that the clients should be distributed in an efficient and common-sense way to the queues. We must simulate this system in our application by entering some parameters that characterize it. The simulation should be easy to understand by a user of the application (that’s why we need a simple, yet intuitive user interface).

The customers are generated randomly, each having its own service time and arrival time. The number of clients to be generated depends on the parameters that the user has chosen.

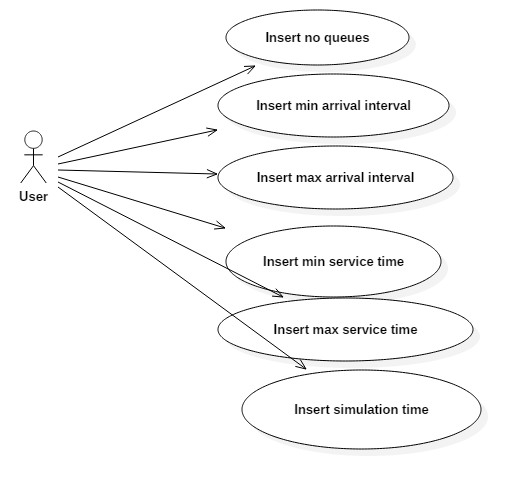
Parameters:

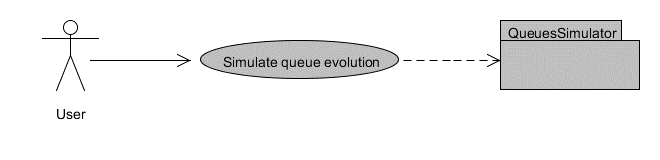
* The maximum number of queues available to process customers. At the beginning, only one queue will be open, and, depending on how many clients are generated and their service time, additional queues will be opened;
* Minimum and maximum arrival interval: the delay between customers arriving to receive a service. When generating clients, the arrival time will be chosen randomly based on this values;
* Minimum and maximum service time: the number of minutes needed for a client to be processed, a value is chosen randomly;
* Simulation time: how much the simulation lasts.

Output shown to the user

* A description of what happened on the queue, all the clients that were there and their arrival time and process time
* The average service time of the served customers;
* The average waiting time of the served customers: how much the customers have waited in queue to receive the service;
* The total service time of all served customers;
* The total waiting time of all served customers;
* The “peak hour”, when the most clients were served;

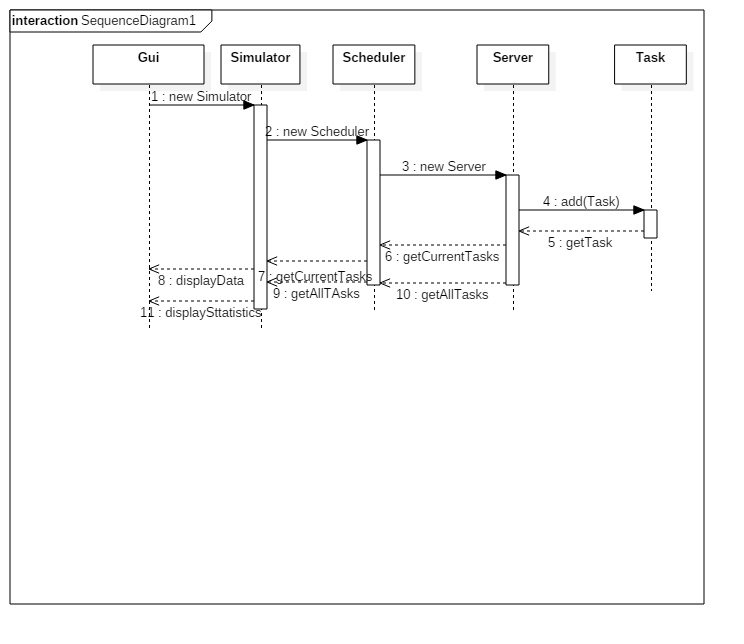
## Scenarios,use cases



****

## Activity diagram

## Sequence diagram



# Results

**Resume**: After inserting the input parameters in the user interface (queue threshold, min. and max. serving time, min. and max. arrival interval, and the simulation time) in the lower part of the window and then pressing the start button, the user starts seeing in real time the simulation of the queuing system. After all the clients have arrived and have been processed, the user can click on the statistics button and see the statistics corresponding to each queue.

**Actors**: User

**Scenarios**:

1. Preconditions: application is ready to use
2. Normal scenario:
3. User successfully inserts the input parameters;
4. User pushes the START button in the left lower part of the window to start the simulation;
5. The application displays in real time the simulation – the clients and their information when they arrive at the queue;
6. The STATISTICS button can now be pressed and the user can see the statistics coresponding to each queue and other additional data.
7. Alternative scenarios(for further development)
   1. 1) User types wrong data in the input fields;

2) User pushes the START button attempting to start the simulation;

3) An error message is displayed;

* 1. 1) User presses the statistics button before the simulation

2) An error message is displayed;

10 10 101 01 01 01 01 10 10 10 101 01 01 10 10 10 10 101 01 10 10 10 10 101 01 01 10 10 10 10 10 10 10 10 10 10 10 101 01 01 10 10 10 10 101 01 10 10 10 101 01 01 10 10 10 10 10 101 01 10 10 101 01 10 01 10 10 10 101010 10100 0 0 0 0 00 0 0 0 00 0 0 0 00 0 0 0 0 0 00 0 0 0 0 0 0 00 0 0 0 0 0 0 0 0 0 0 0 0 00 0 0 35 34 21 4 132 423 42 13 412 4 213 4321 4 1234 213 42 134 12 423 4 21 4 12 421 34 1234 21 42 14 21 341 234 21 412 4 214 213 4231 4 21 4213 4 1234 12 4 21 4231 41 231 412 4 123 4213 4 12 412 34 123 4 213 421 34 123 4 135 3 426 3645 764 58 76 856 57 8 547 35 2 2 4 1 2413 4 2 53245 346 2 6 5 21 3 21 51 435 32 5 23 45423 6 326 46 425 12 43 1324 5 12345 234 5 432534 5 42 54 5 5 5 5 5 5 55 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

10 10 101 01 01 01 01 10 10 10 101 01 01 10 10 10 10 101 01 10 10 10 10 101 01 01 10 10 10 10 10 10 10 10 10 10 10 101 01 01 10 10 10 10 101 01 10 10 10 101 01 01 10 10 10 10 10 101 01 10 10 101 01 10 01 10 10 10 101010 10100 0 0 0 0 00 0 0 0 00 0 0 0 00 0 0 0 0 0 00 0 0 0 0 0 0 00 0 0 0 0 0 0 0 0 0 0 0 0 00 0 0 35 34 21 4 132 423 42 13 412 4 213 4321 4 1234 213 42 134 12 423 4 21 4 12 421 34 1234 21 42 14 21 341 234 21 412 4 214 213 4231 4 21 4213 4 1234 12 4 21 4231 41 231 412 4 123 4213 4 12 412 34 123 4 213 421 34 123 4 135 3 426 3645 764 58 76 856 57 8 547 35 2 2 4 1 2413 4 2 53245 346 2 6 5 21 3 21 51 435 32 5 23 45423 6 326 46 425 12 43 1324 5 12345 234 5 432534 5 42 54 5 5 5 5 5 5 55 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

# Design-UML Diagram

# Class Design

**Server** represents a queue from the real world, where customers wait in line to receive a service. Tasks (which could be seen as the customers) can be assigned to a queue by using the method dispatchTaskOnServer*();*

**Task** is representing the client or the element sitting at the queue and has two parameters(arrival time, service time). Tasks are generated randomly during the simulation, and assigned to the queue which would be the best for them (the sum of the processing times of the tasks that are already in the queue is the smallest)

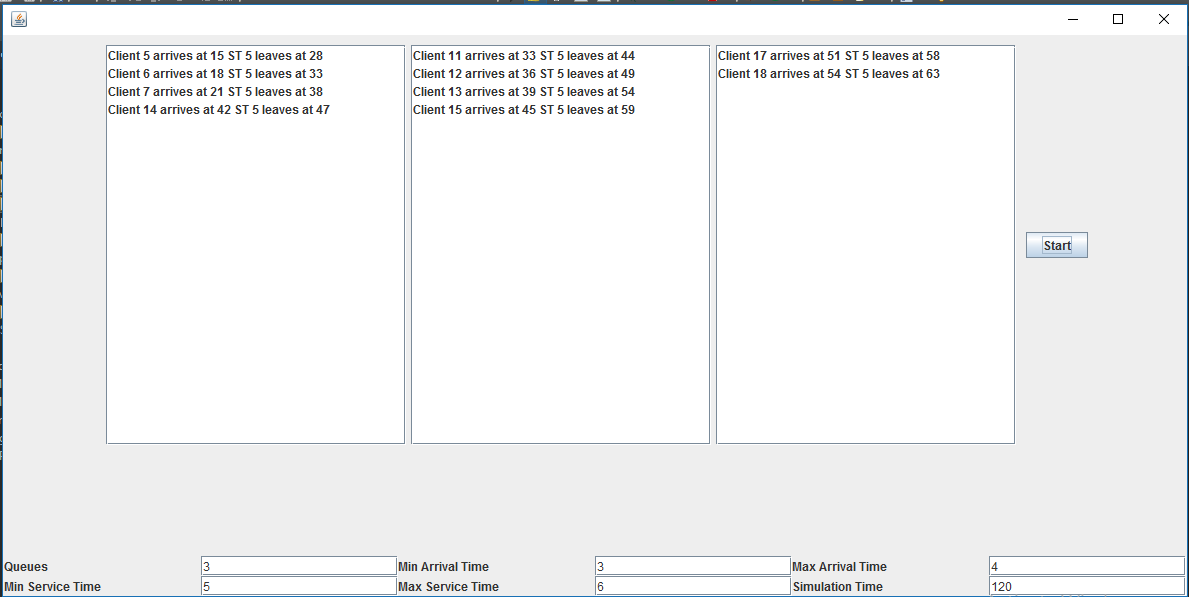
**Scheduler** is the class where we have our queues – an array of queues. Here we keep track of the best queue to put a new task at, we also have the place where we dispatch the task on the queue (that fits best).

**Simulator** is the class where we do the actual simulation, where we create tasks and generate parameters for each task, where we dispatch them on queues and show the queues to the viewer by declaring a simulator frame, that is presented below. Here is where most of the logic of the program resides, where all the computations and statistics are made. This ties everything together.

**Simulator frame** is the class that represents the simulation saw by the viewer. It contains a grid proportional to the number of queues and shows all the queues, also the input boxes and the buttons.

**Statistics frame** is the class that corresponds to the view where the user can see the statistics after the simulation.

# Graphical User Interface-GUI



The Graphical User Interface (GUI) is meant to be user-friendly and easy to use, even for non-specialists, due to the intuitive controls. When starting the application a new window will open.

After the user inputs valid data for the required input fields, presses the START button and the simulation will take place and the user sees in real time what happens on the queues.

After the simulation has ended, the user can see all the information he needs about the queues. He can see all the tasks that were assigned to each queue, their corresponding arrival times and processing times, the average waiting time per task per queue, the average processing time per task per queue, the empty queue time for each queue, the number of clients (tasks) that were at a queue for each hour in the simulation time and the peak hour.

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

# Implementation

The application has been implemented in Java Programming language, using Eclipse Java Mars an an IDE. For the GUI, I used the Swing library in Eclipse. All interface elements are created in layouts, by code, not with a drag and drop editor, in an efficient way, and because of using grid layouts they addapt to each screen resolution nicely. Listeners are placed for every component to catch specific events (usually a simple button press), and respond accordingly.

On the logical part, some of the methods were a bit more difficult than others. Here, I chose to exemplify with some of them.

The main new element in this project was using threads. Threads were used because they allow different queues to be working concurrently and simulated just like they would happen in real life.

With a threads package, a programmer can create multiple threads within a process. Threads execute concurrently and, within a multithreaded process, there are at any time multiple points of execution. Threads execute within a single address space. Multithreaded programming offers the following advantages:

· **Performance**

Threads improve the performance (throughput, computational speed, responsiveness, or some combination of these) of a program. Multiple threads are useful in a multiprocessor system where threads run concurrently on separate processors. In addition, multiple threads also improve program performance on single processor systems by permitting the overlap of input and output or other slow operations with computational operations.

You can think of threads as executing simultaneously, regardless of the number of processors present. You cannot make any assumptions about the start or finish times of threads or the sequence in which they execute, unless explicitly synchronized.

**· Shared Resources**

An advantage of using multiple threads over using separate processes is that the former share a single address space, all open files, and other resources.

**· Potential Simplicity**

Multiple threads can reduce the complexity of some applications that are inherently suited for threads.

# Testing

* 1. **Testing**

For a given input of:

Queue threshold: 5

Maximum number of queues: 3

Minimum arrival time: 2

Maximum arrival time: 3

Minimum service time: 2

Maximum service time: 6

Simulation time: 300

The displayed output is:

Queue #1

Arival time: 5 Process time: 4

Arival time: 8 Process time: 4

Arival time: 10 Process time: 2

Arival time: 14 Process time: 4

Arival time: 18 Process time: 4

Arival time: 23 Process time: 2

Arival time: 26 Process time: 3

Arival time: 53 Process time: 3

Arival time: 58 Process time: 4

Arival time: 69 Process time: 2

Arival time: 71 Process time: 4

Arival time: 92 Process time: 3

Arival time: 107 Process time: 4

Arival time: 109 Process time: 2

Arival time: 117 Process time: 2

Arival time: 126 Process time: 3

Arival time: 136 Process time: 2

Arival time: 143 Process time: 3

Arival time: 146 Process time: 4

Arival time: 157 Process time: 4

Arival time: 177 Process time: 3

Arival time: 183 Process time: 3

Arival time: 190 Process time: 4

Arival time: 203 Process time: 3

Arival time: 211 Process time: 2

Arival time: 220 Process time: 4

Arival time: 229 Process time: 2

Arival time: 232 Process time: 4

Arival time: 250 Process time: 2

Arival time: 253 Process time: 4

Arival time: 264 Process time: 4

Arival time: 270 Process time: 2

Arival time: 281 Process time: 5

Arival time: 294 Process time: 5

Average waiting time: 14.12

Average processing time: 3.24

Empty queue time: 480

Queue #2

Arival time: 29 Process time: 4

Arival time: 33 Process time: 2

Arival time: 38 Process time: 2

Arival time: 43 Process time: 5

Arival time: 45 Process time: 3

Arival time: 49 Process time: 3

Arival time: 62 Process time: 2

Arival time: 67 Process time: 4

Arival time: 73 Process time: 2

Arival time: 78 Process time: 4

Arival time: 101 Process time: 5

Arival time: 112 Process time: 3

Arival time: 122 Process time: 5

Arival time: 133 Process time: 4

Arival time: 148 Process time: 4

Arival time: 160 Process time: 2

Arival time: 169 Process time: 2

Arival time: 179 Process time: 3

Arival time: 186 Process time: 2

Arival time: 193 Process time: 2

Arival time: 199 Process time: 4

Arival time: 206 Process time: 4

Arival time: 215 Process time: 3

Arival time: 225 Process time: 5

Arival time: 245 Process time: 5

Arival time: 261 Process time: 5

Arival time: 275 Process time: 2

Arival time: 278 Process time: 4

Arival time: 288 Process time: 4

Arival time: 298 Process time: 4

Average waiting time: 13.23

Average processing time: 3.43

Empty queue time: 397

Queue #3

Arival time: 80 Process time: 5

Arival time: 83 Process time: 2

Arival time: 86 Process time: 3

Arival time: 89 Process time: 3

Arival time: 96 Process time: 3

Arival time: 98 Process time: 2

Arival time: 104 Process time: 5

Arival time: 120 Process time: 4

Arival time: 131 Process time: 3

Arival time: 139 Process time: 4

Arival time: 153 Process time: 3

Arival time: 165 Process time: 5

Arival time: 173 Process time: 5

Arival time: 195 Process time: 5

Arival time: 201 Process time: 3

Arival time: 218 Process time: 5

Arival time: 237 Process time: 4

Arival time: 241 Process time: 5

Arival time: 257 Process time: 5

Arival time: 266 Process time: 3

Arival time: 286 Process time: 5

Arival time: 291 Process time: 5

Arival time: 296 Process time: 5

Average waiting time: 12.78

Average processing time: 4

Empty queue time: 294

**Peak hour: 2**

# What I have learned

I have learned that dealing with threads is more complicated that I have previously expected.In my initial version of the project,I have failed into resolving the errors given by multiple threads trying to modify a variable,so I had to begin it from scratch again.I have learned how to catch an exception thrown by threads and memory consistency errors.For the later I have used a new type of integer,the AtomicInteger,which heled me avoid errors and maintain synchronization.

When designing the classes that contained threads,I had to implement either Runnable or Thread. As we know we can implement thread either by extending Thread class or implementing Runnable interface, the question arise, which one is better and when to use one? This question will be easy to answer if you know that Java programming language doesn't support multiple inheritances of class, but it allows you to implement multiple interfaces. Which means, it's better to implement Runnable.

Then came the question of what methods should be overridden. One of trick Java question from early days, but still good enough to differentiate between shallow understanding of Java threading model start() method is used to start newly created thread, while start() internally calls run() method, there is difference calling run() method directly. When you invoke run() as normal method, its called in the same thread, no new thread is started, which is the case when you call start() method.